UM EECS 270 F22
Introduction to Logic Design

15. Register Transfer Level (RTL) Design

Data vs. Control State

- Not all bits are equal!
- Some bits have meaning only in the context of other bits, e.g.,
  - Positional weight (numbers)
  - Codes (ASCII, Unicode, Colors, …)
- Multi-bit "words" represent data → 2's complement numbers & pixel ...
- Single bits represent control → How the data moves around
- Data state (lots of bits) >> control state (few bits) (think 64-bit numbers) in the large digital systems
- Encoding sequential circuits that involve data at the single-bit level:
  - Leads to the so-called state explosion, → think about a sequential circuit that has 100,000 flipflops that's 100,000 bits of states → the number of possible state: $2^{100,000}$
  - Loses the semantics (meaning) of data, and is lose the high-level semantic
  - Unnecessary

Deal with state explosion: By raising the level of abstraction of how we describe these much larger digital systems.

We start to distinguish between different kinds of bits / different kinds of structures in our digital system.

Register Transfer Level (RTL)

- **Datapath**: Computation on "wide" signals
  - Arithmetic (Add, Subtract, Multiply, Count, etc.)
  - Logical (Shift right/left, Arith Shift, bit-wise, etc.)
  - Other (Clear, load, hold)

logical operations on multi bits

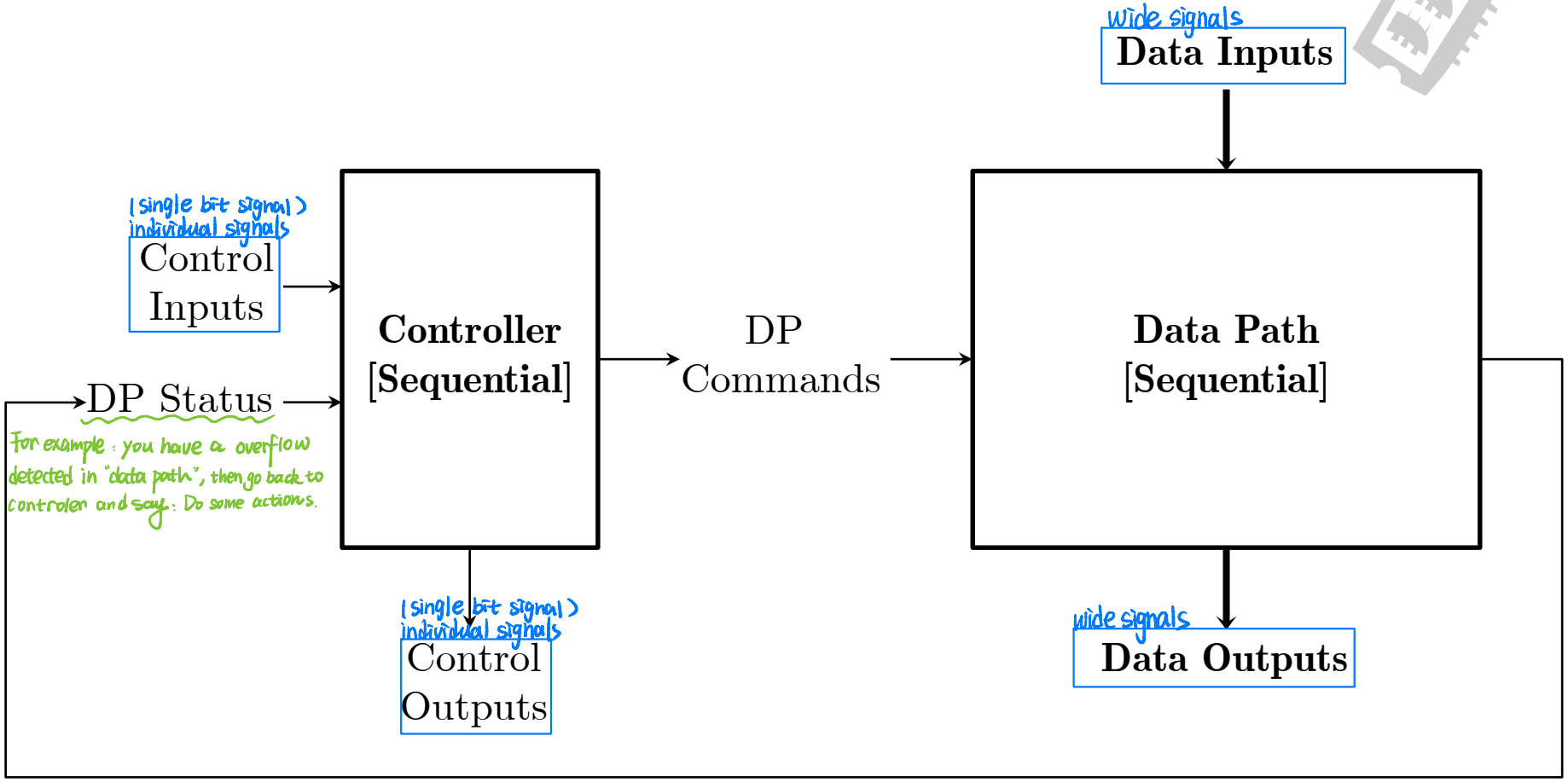
take the whole 64 bit and say let's do a bit wise "or" on the bits of that with some other 64 bit quantity

- **Controller**: Orchestrating Datapath operations

addition or subtraction

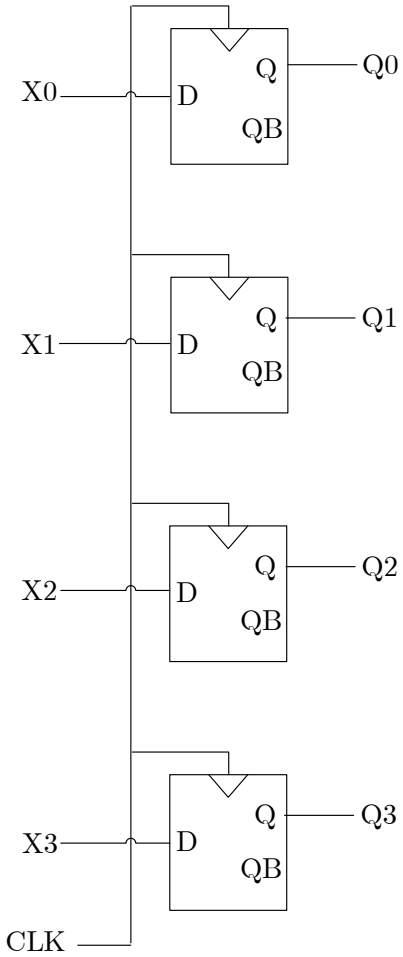
two: Traffic light : Controller
Car : Data.

Structure of RTL Sequential Circuit



it just a bunch of flip flops that are next to each other.

Register State Space



$$Q_0^+ = D_0 = X_0$$
$$Q_1^+ = D_1 = X_1$$
$$Q_2^+ = D_2 = X_2$$
$$Q_3^+ = D_3 = X_3$$

$$\#\text{States} = 2^4 = 16$$

$$\#\text{Transitions} = 2^4 \text{ transitions/state} \times \#\text{States} = 256$$

当 states 很多时，state diagram 就会变得没有意义，因为图中 state 相互连接很多

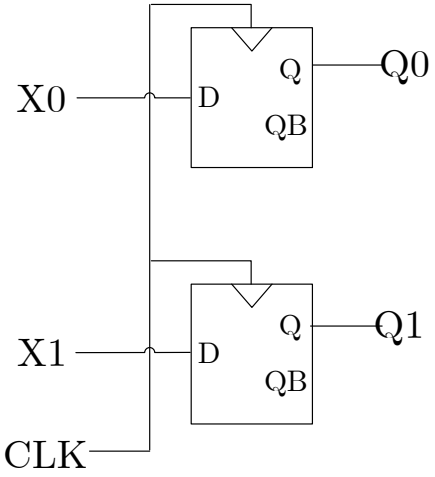
State table and state diagram representations
too *large and unwieldy*

Next-state equations
much *simpler* and very *compact*

更好

on every rising edge of the clock, the x values move in the Q values. ⟶ inputs get stored. $Q^+ = X$ ($Q_1^+ = x_1$, $Q_2^+ = x_2 \cdots$)

Register State Space



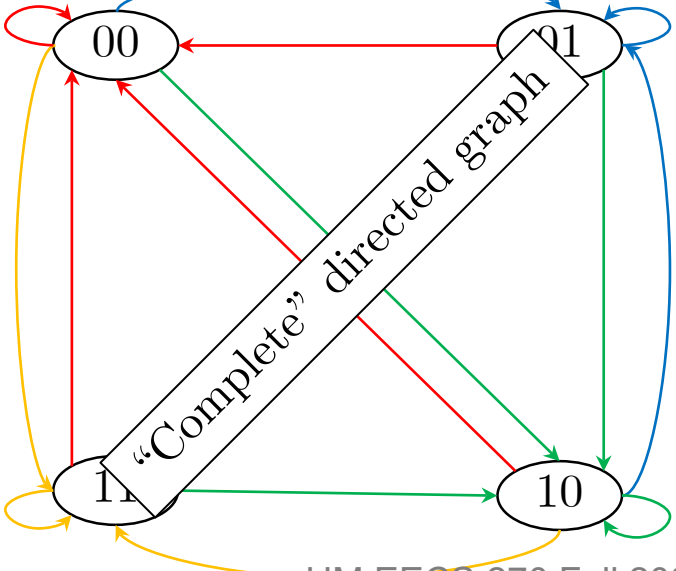
$$Q_0^+ = X_0$$
$$Q_1^+ = X_1$$

$$\#\text{States} = 4$$
$$\#\text{Transitions} = 16$$

Q_1Q_0	X_1X_0			
	00	01	10	11
00	00	01	10	11
01	00	01	10	11
10	00	01	10	11
11	00	01	10	11

$Q_1^+Q_0^+$

every thing is connected.

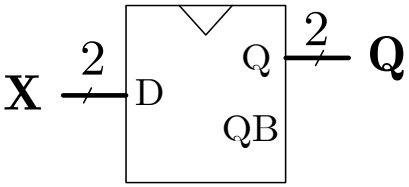


$X_1X_0 = 00$ →
$X_1X_0 = 01$ →
$X_1X_0 = 10$ →
$X_1X_0 = 11$ →

$$\mathbf{Q} \triangleq [Q_1Q_0]$$
$$\mathbf{X} \triangleq [X_1X_0]$$

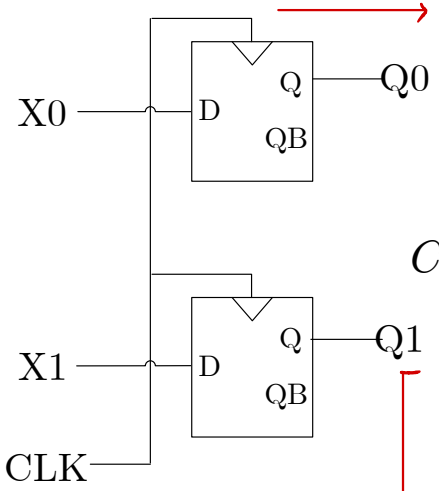
expree as vector

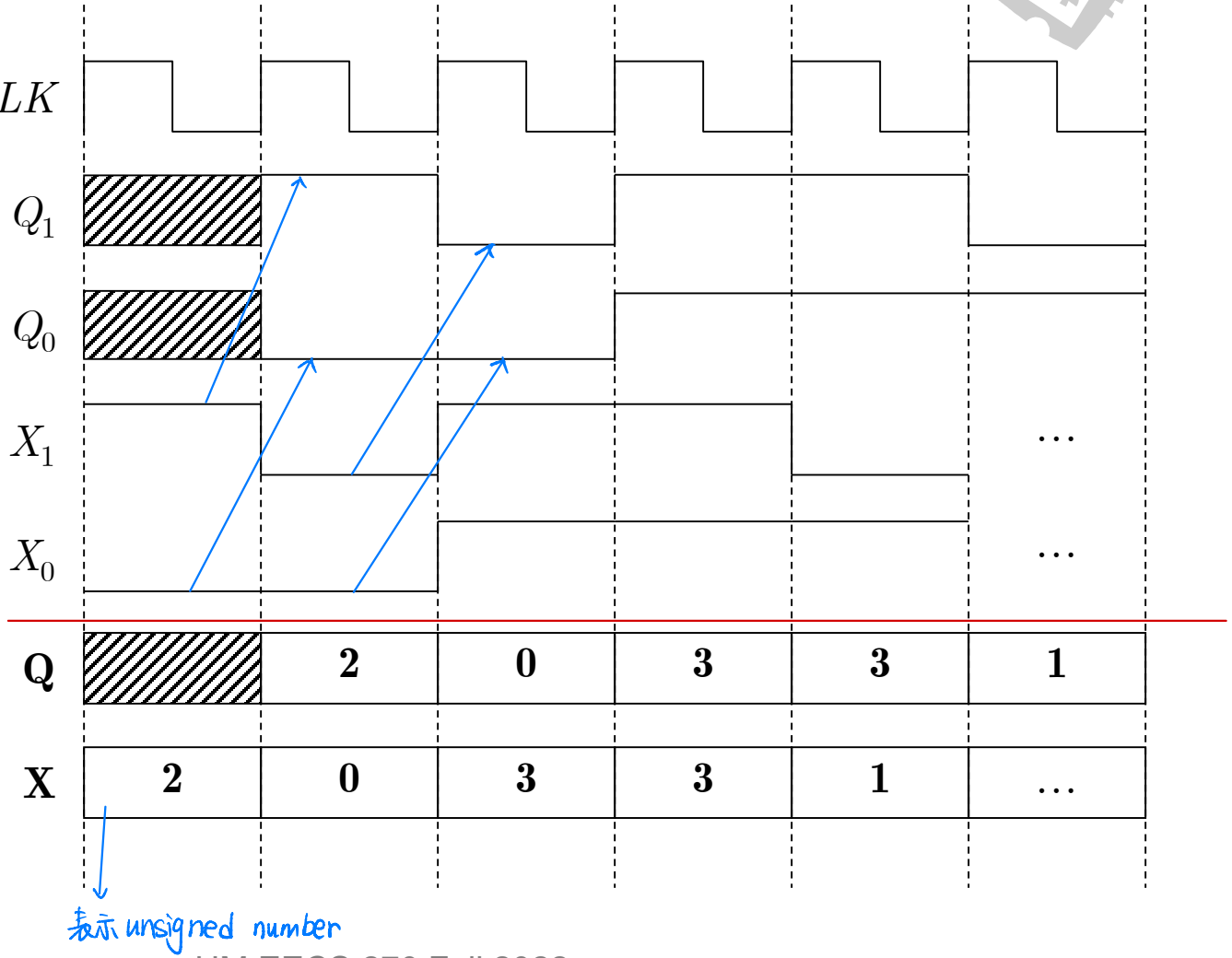
$$\mathbf{Q}^+ = \mathbf{X}$$



Register Timing

this is a positive edge triggered register





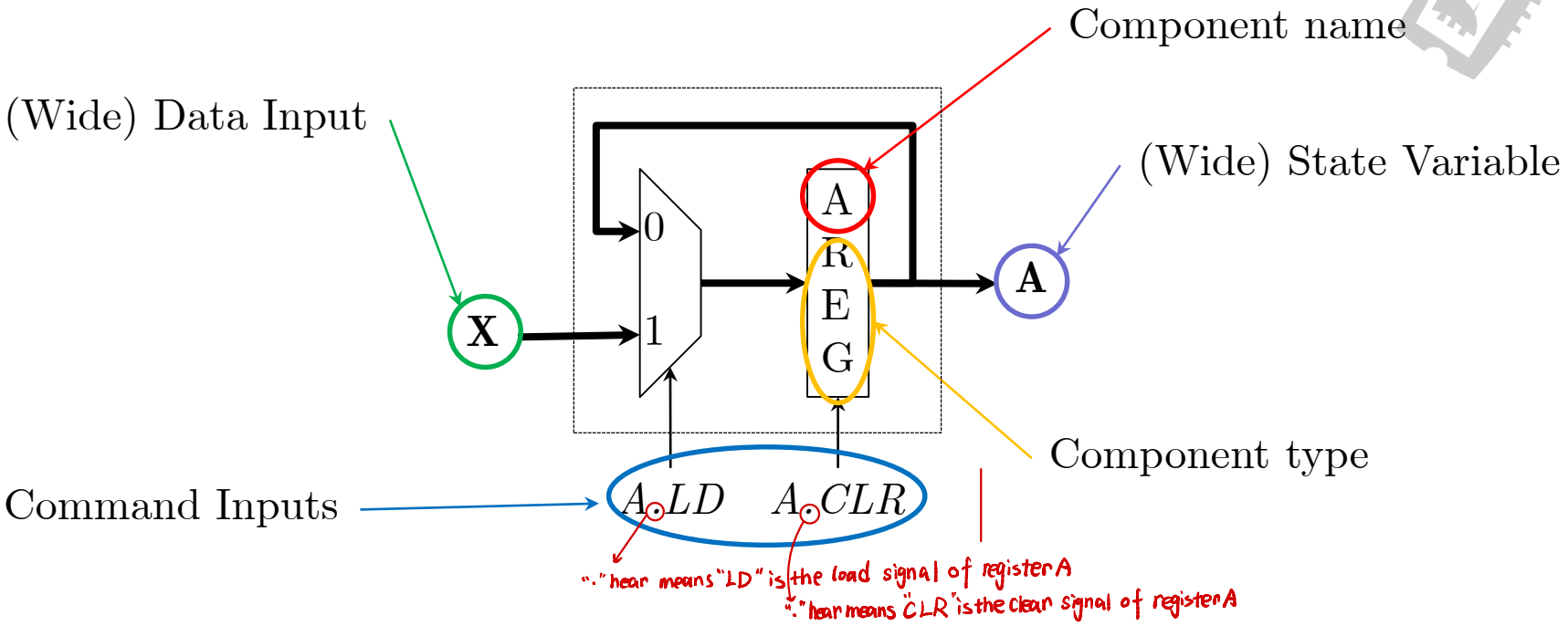
we can summarize this, instead of writing all the Q and X.

改进

interpret the input and output as a number

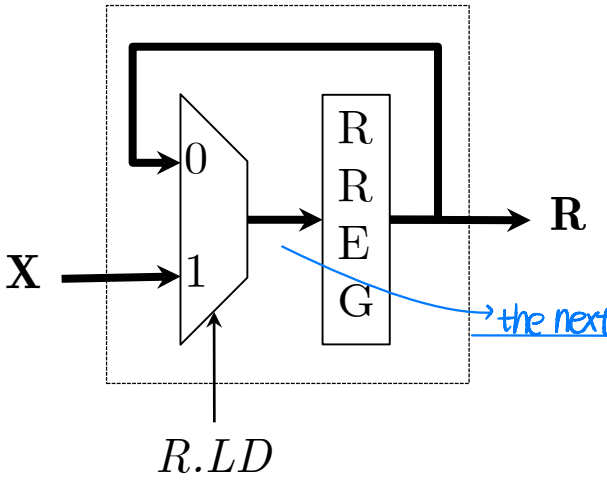
表示 unsigned number

Register with Load & Clear Inputs



$$\mathbf{A}^{+} = A.CLR\,?\,\mathbf{0} : A.LD\,?\,\mathbf{X} : \mathbf{A}$$

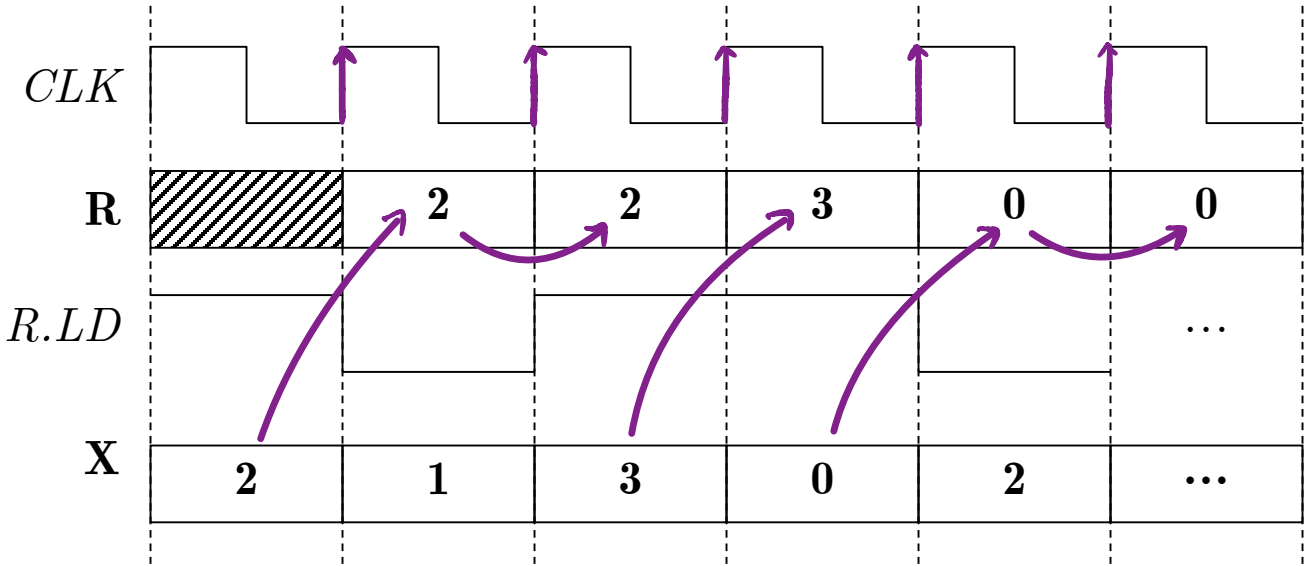
Timing: Register R with Load



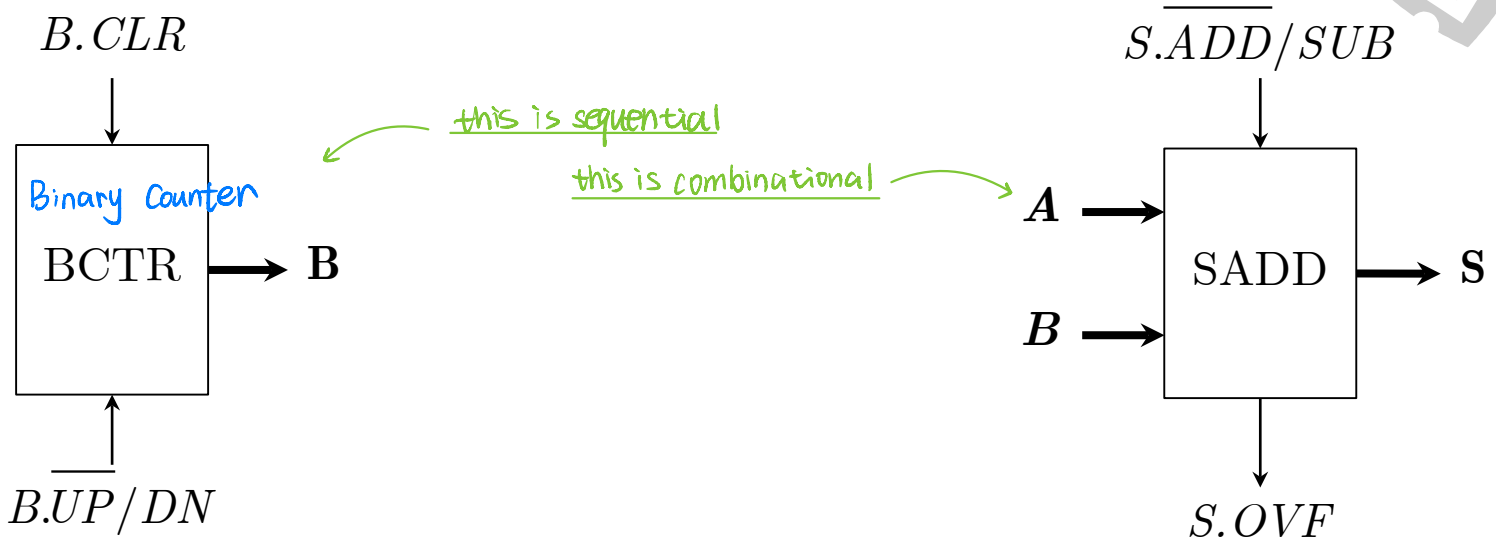
R^+ → the next state of R depends on ① wide input X, ② control signal R.LD, ③ previous state R

there is no initialization, just loading → loading a new value; keep the old value

$$\mathbf{R}^+ = R.LD\,?\,\mathbf{X} : \mathbf{R}$$



Other Datapath Components



$$\mathbf{B}^{+} = B.CLR\,?\,\mathbf{0} : B.\overline{UP}/DN\,?\,\mathbf{B}-1 : \mathbf{B}+1$$

$$\mathbf{S} = S.\overline{ADD}/SUB\,?\,\mathbf{A}-\mathbf{B} : \mathbf{A}+\mathbf{B}$$

Other Datapath Operations

- Shifting (right, left, arithmetic)
- Rotating (right, left)
- Bit-field extraction/concatenation
- What else?

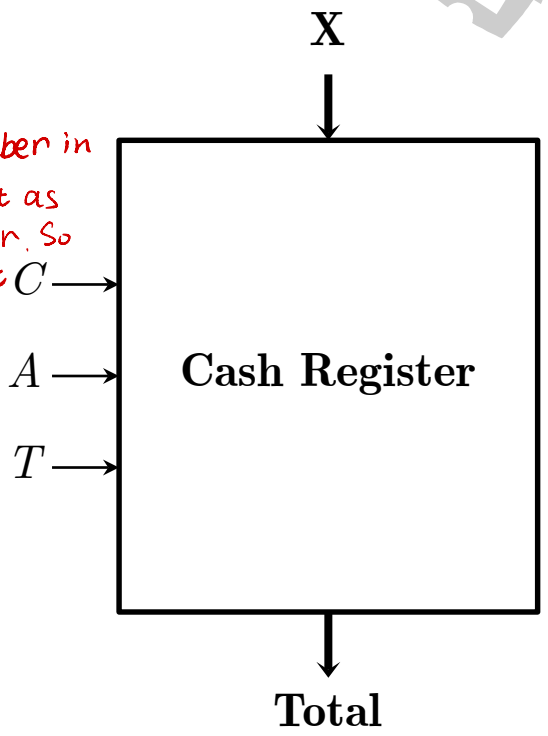
A "Cash Register"

- Function:
  Compute total price of a number of items
- Inputs:
  - **X**: W-bit unsigned integer
  - C: Clear
  - A: Add
  - T: Total
- Output:
  - **Total**: W-bit sum of added numbers
- Spec:
  - $C = 1$ clears **Total**
  - $A = 1$ adds next **X**
  - $T = 1$ displays **Total**
  - C, A, and T are one-hot

you dont need to specify the number of bits like a fixed number in the code. You can specify it as variable W or a parameter. So you can change the width without change other thing.

also have zero-hot. eg: "1011"

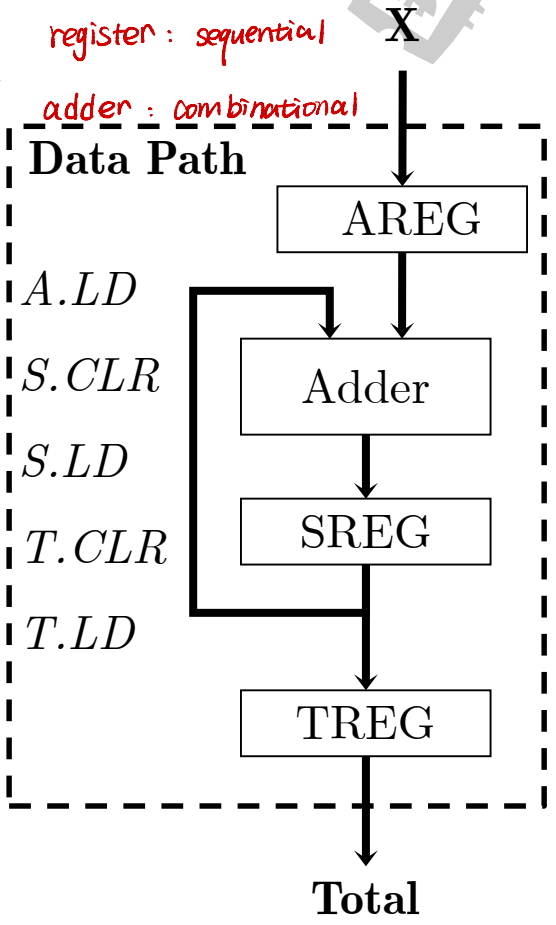
meaning: We have 3 control signals. → eight combination. However, only 4 of them are valid:



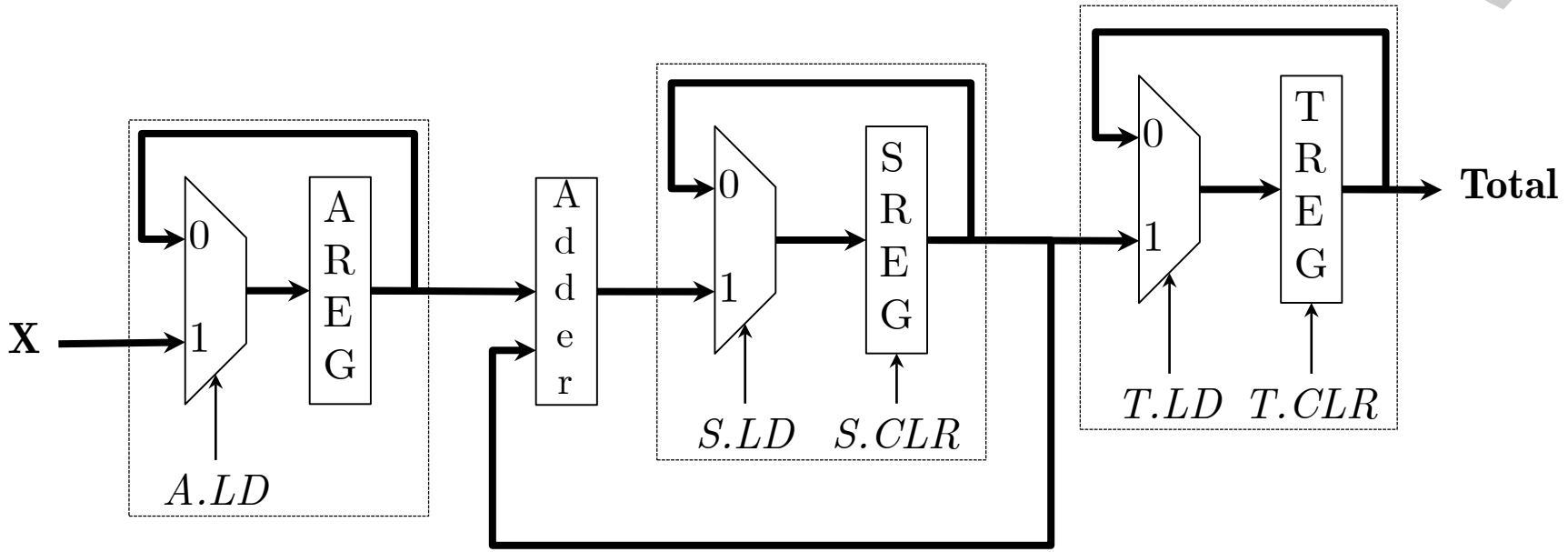
Designing the Datapath

- Required Components: Figure out what do you want to store what operation do you need to do.
  - Register to store input **X**: AREG
  - Register to store output **Total**: TREG
  - Register to store intermediate sums: SREG
  - Adder
- Component functionality:
  - AREG: Load ($A.LD$)
  - TREG: Load ($T.LD$) and Clear ($T.CLR$)
  - SREG: Load ($S.LD$) and Clear ($S.CLR$)
- Datapath Architecture:

because between different customers I need to reset the total register and the sum register to "0"

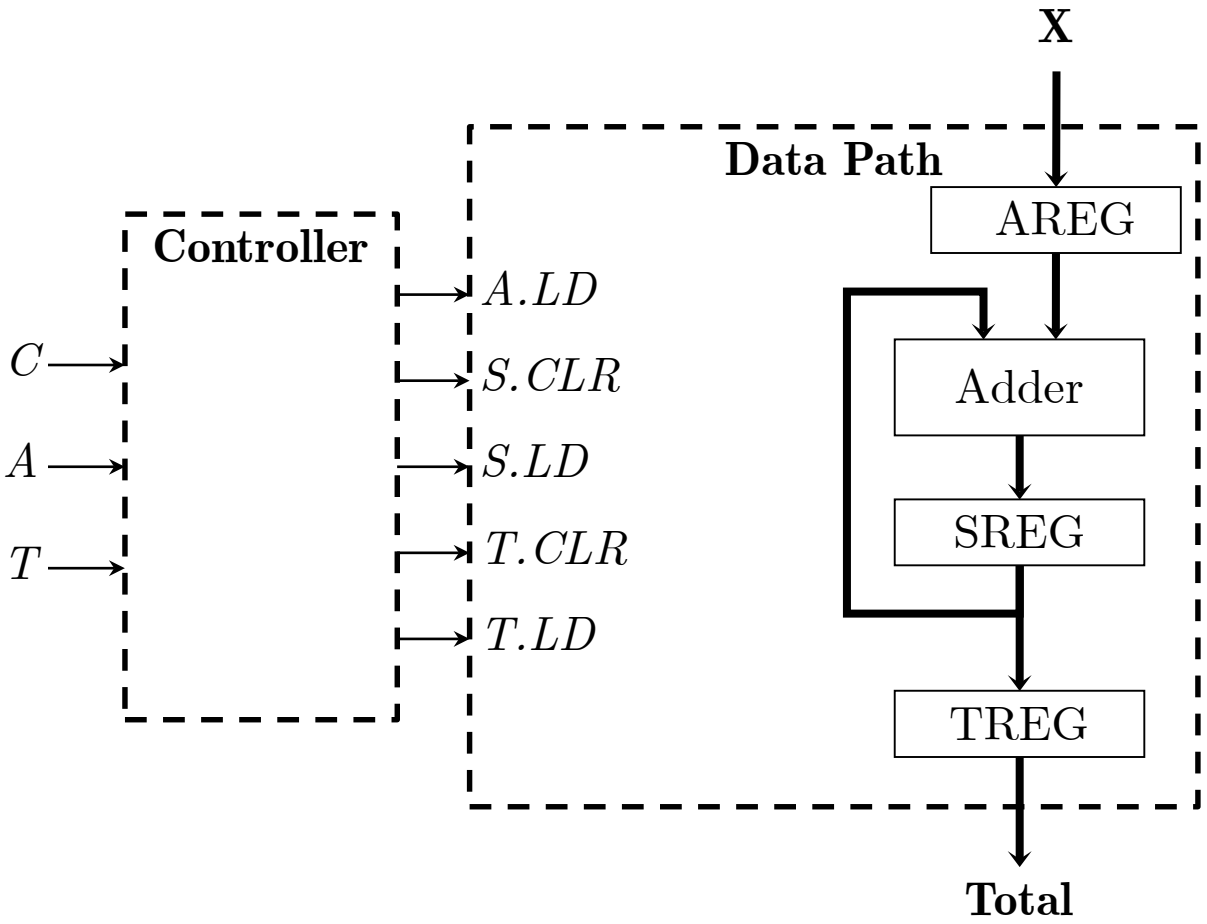


Detailed Datapath

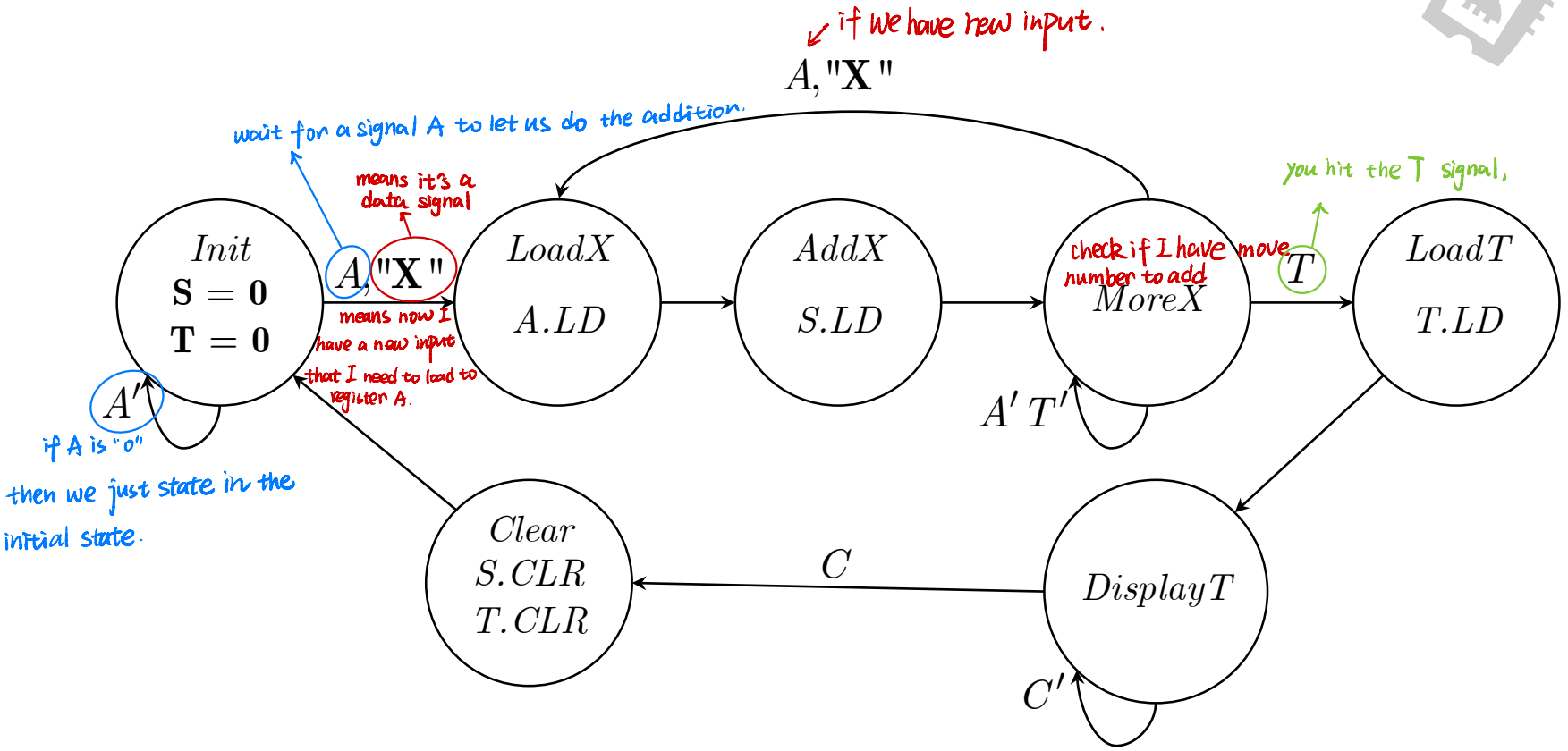


$$\mathbf{A}^{+} = A.LD\,?\,\mathbf{X} : \mathbf{A} \qquad \mathbf{S}^{+} = S.CLR\,?\,\mathbf{0} : \Big(S.LD\,?\,\big(\mathbf{S}+\mathbf{A}\big) : \mathbf{S}\Big) \qquad \mathbf{T}^{+} = T.CLR\,?\,\mathbf{0} : \Big(T.LD\,?\,\mathbf{S} : \mathbf{T}\Big)$$

Designing the Controller



Controller State Diagram



```
module CashReg
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller State Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module CashReg
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller State Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

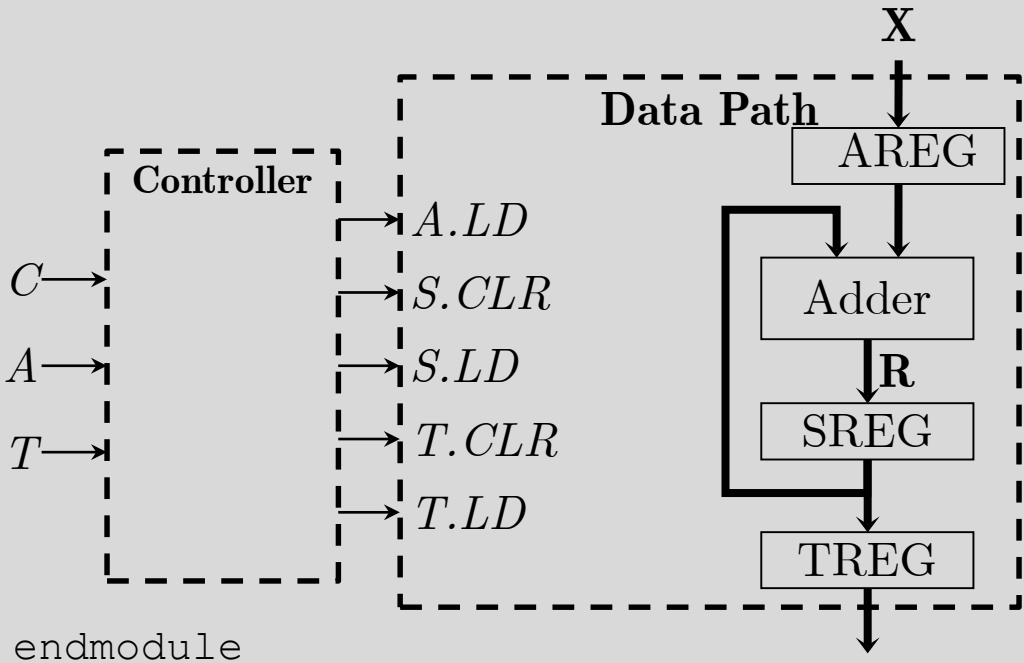
```
module CashReg
 #(parameter W = 10)
 (
   input Clock,
   input A,
   input T,
   input C,

   input [W-1:0] X,
   output [W-1:0] Total,
   output [2:0] QState
 );

endmodule
```

default. when you instanciate this, you can change the W here.

control input

W bit

用于simulation, 因为我们有7个state, 那么就预留3个Qstate.



```
module CashReg
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller State Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module CashReg
 reg [W-1:0] AREG;
 wire A_LD;

 reg [W-1:0] SREG;
 wire S_CLR, S_LD;

 reg [W-1:0] TREG;
 wire T_CLR, T_LD;

 wire ovf;
 wire [W-1:0] R;
 AddW #(.W(W))
   Adder(SREG, AREG, 1'b0, R, ovf);
endmodule
```

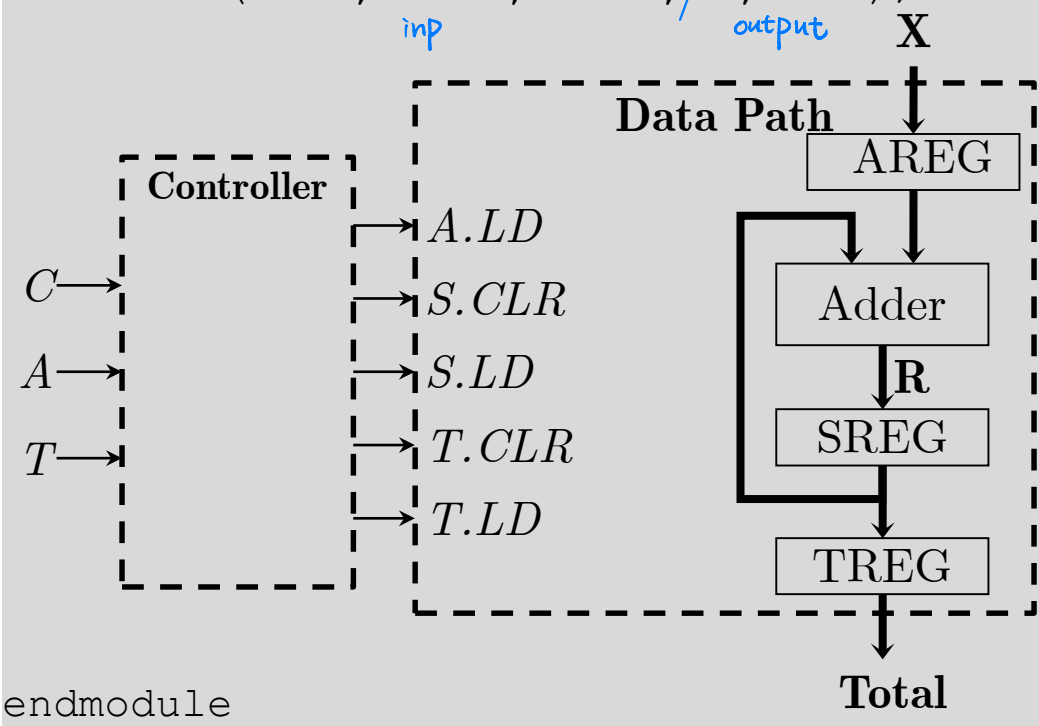
controlers

for every datapath component you just say: Here is the register, Here is the controler that will affect register next state.

carry in

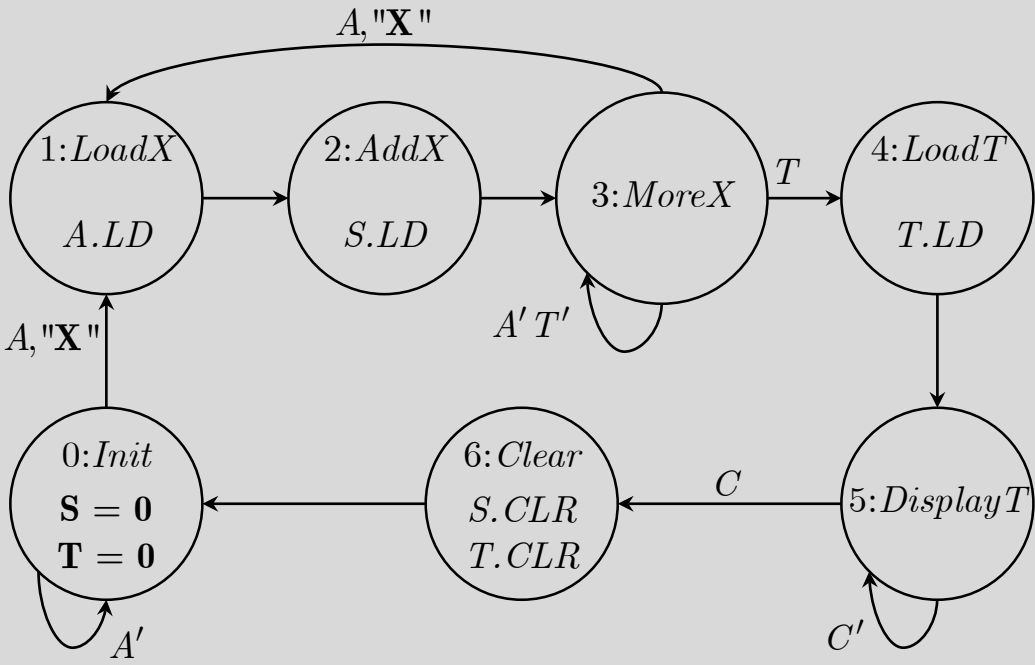
inp

output



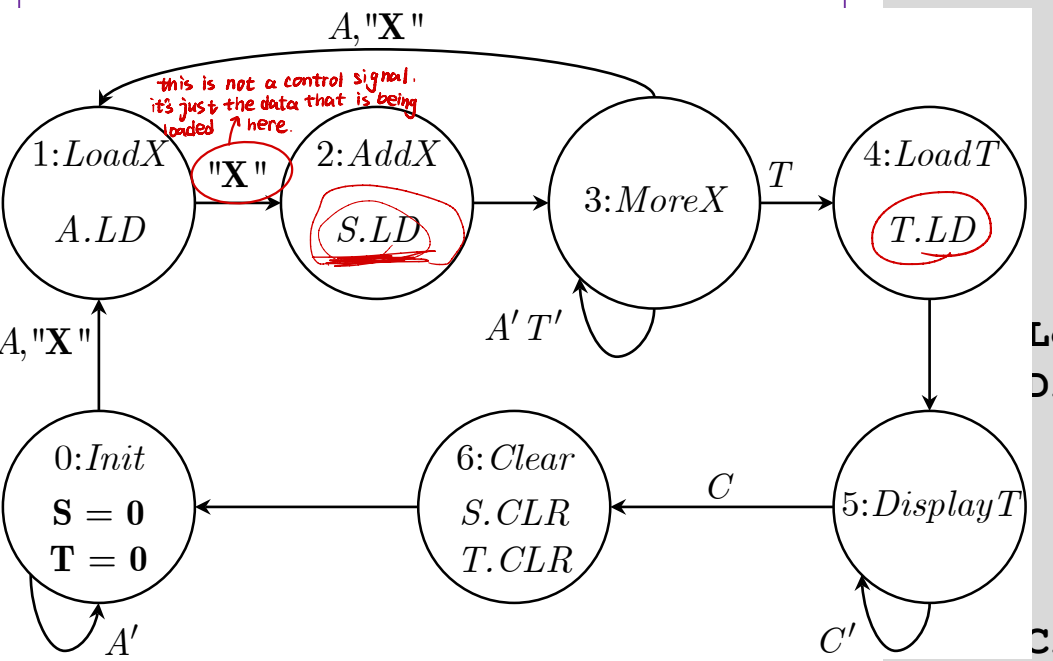
```
module CashReg
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller State Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module CashReg
 reg [2:0] Q, Q_Next;
 localparam Init          = 3'd0;
 localparam LoadX         = 3'd1;
 localparam AddX          = 3'd2;
 localparam MoreX         = 3'd3;
 localparam LoadTotal     = 3'd4;
 localparam DisplayTotal  = 3'd5;
 localparam Clear         = 3'd6;
```




```
endmodule
```

```
module CashReg
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller State Transitions
```




```
module CashReg
always @*
 begin
   case (Q)
     Init:
       if (A)
         Q_Next <= LoadX;
       else
         Q_Next <= Init;
     LoadX: Q_Next <= AddX;
     AddX: Q_Next <= MoreX;
     MoreX:
       if (A)
         Q_Next <= LoadX;
       else if (T)
         Q_Next <= LoadTotal;
       else
         Q_Next <= MoreX;
     LoadTotal: Q_Next <= DisplayTotal;
     DisplayTotal:
       if (C)
         Q_Next <= Clear;
       else
         Q_Next <= DisplayTotal;
     Clear: Q_Next <= Init;
   endcase
 end
endmodule
```

```
module CashReg
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller State Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module CashReg
 initial
 begin
   Q <= Init;
   AREG <= 'd0;
   SREG <= 'd0;
   TREG <= 'd0;
 end

endmodule
```

```
module CashReg
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller State Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

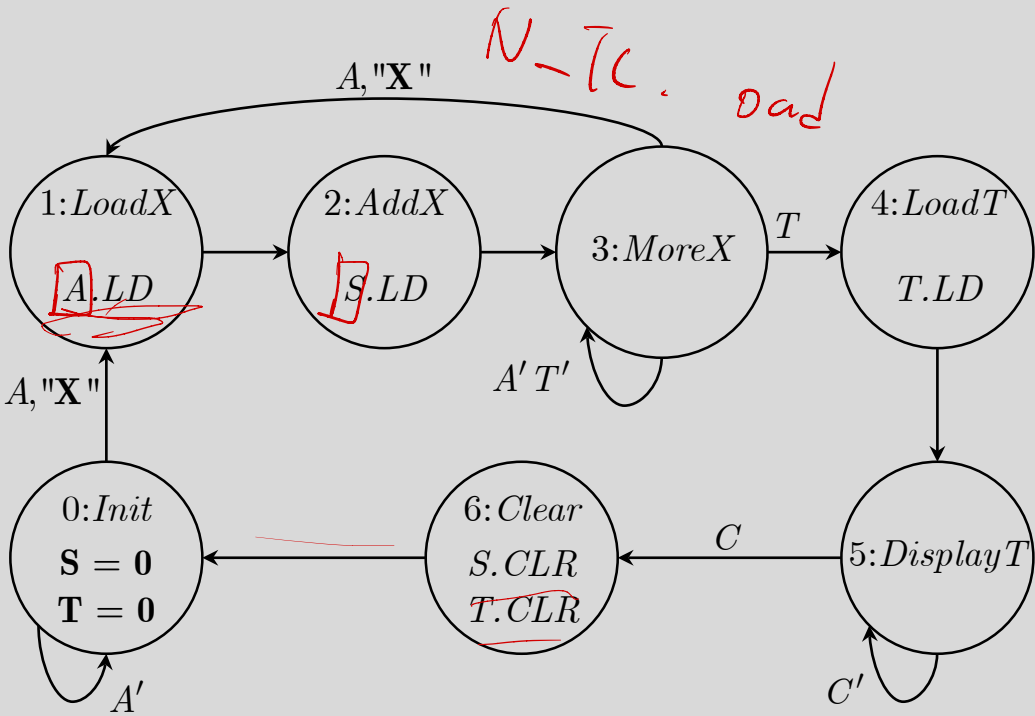
```
module CashReg
 always @(posedge Clock)
   Q <= Q_Next;
```

Q here is actually a 3 bit signal

```
endmodule
```

```
module CashReg
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller State Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

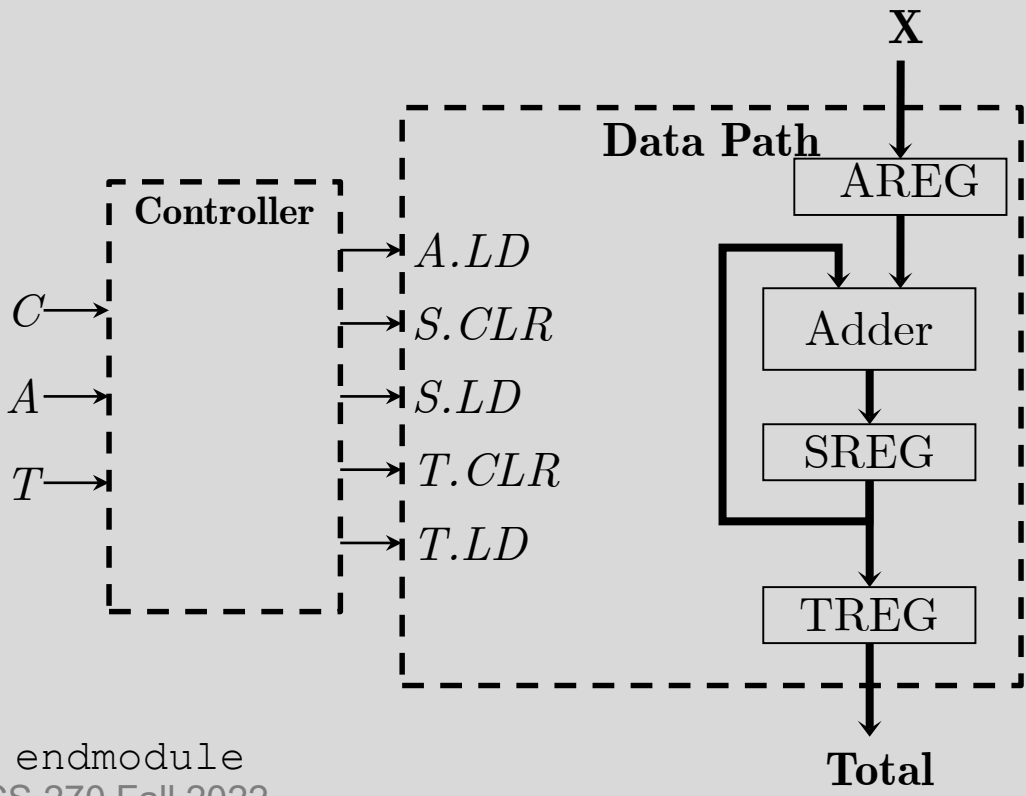
```
module CashReg
 assign A_LD  = (Q == LoadX);
 assign S_LD  = (Q == AddX);
 assign T_LD  = (Q == LoadTotal);
 assign S_CLR = (Q == Clear);
 assign T_CLR = (Q == Clear);
```




```
endmodule
```

```
module CashReg
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller State Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module CashReg
 always @(posedge Clock)
 begin
   AREG <= A_LD ? X : AREG;
   SREG <=
     S_CLR ? 'd0 :
       (S_LD ? R : SREG);
   TREG <=
     T_CLR ? 'd0 :
       (T_LD ? SREG : TREG);
 end
```

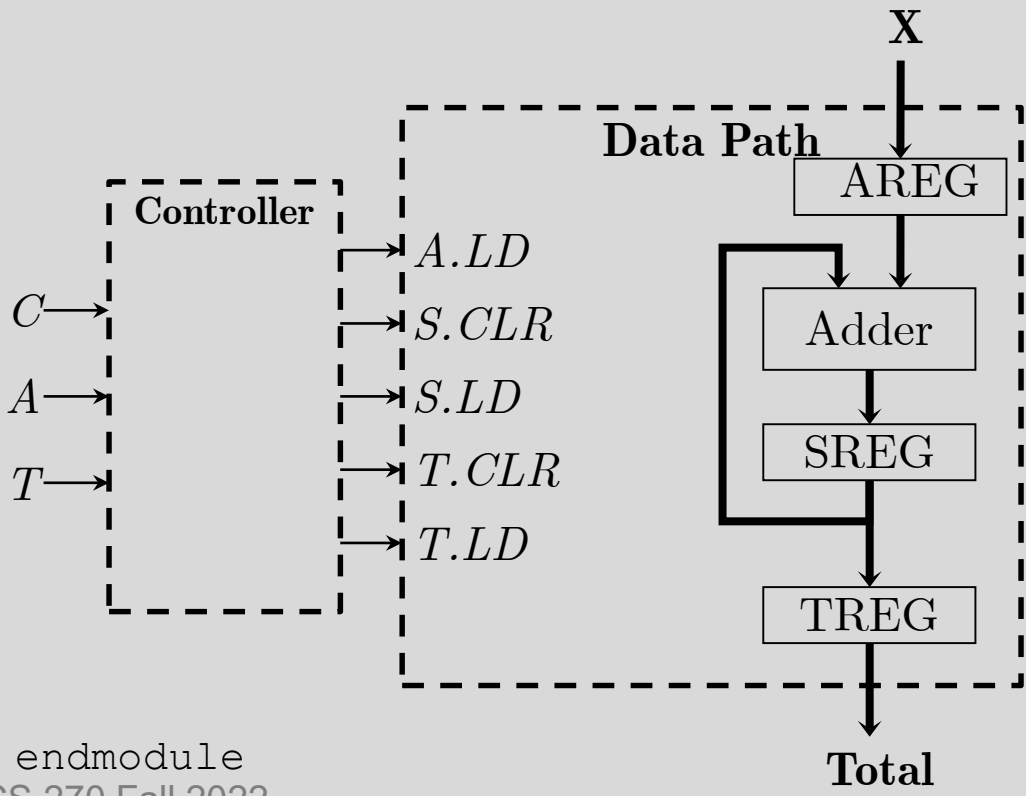



```
endmodule
```

```
module CashReg
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller State Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module CashReg
 always @(posedge Clock)
 begin
   AREG <= A_LD ? X : AREG; // WRONG
   SREG <=
     S_CLR ? 'd0 :
       (S_LD ? R : SREG);
   TREG <=
     T_CLR ? 'd0 :
       (T_LD ? SREG : TREG);
 end
```

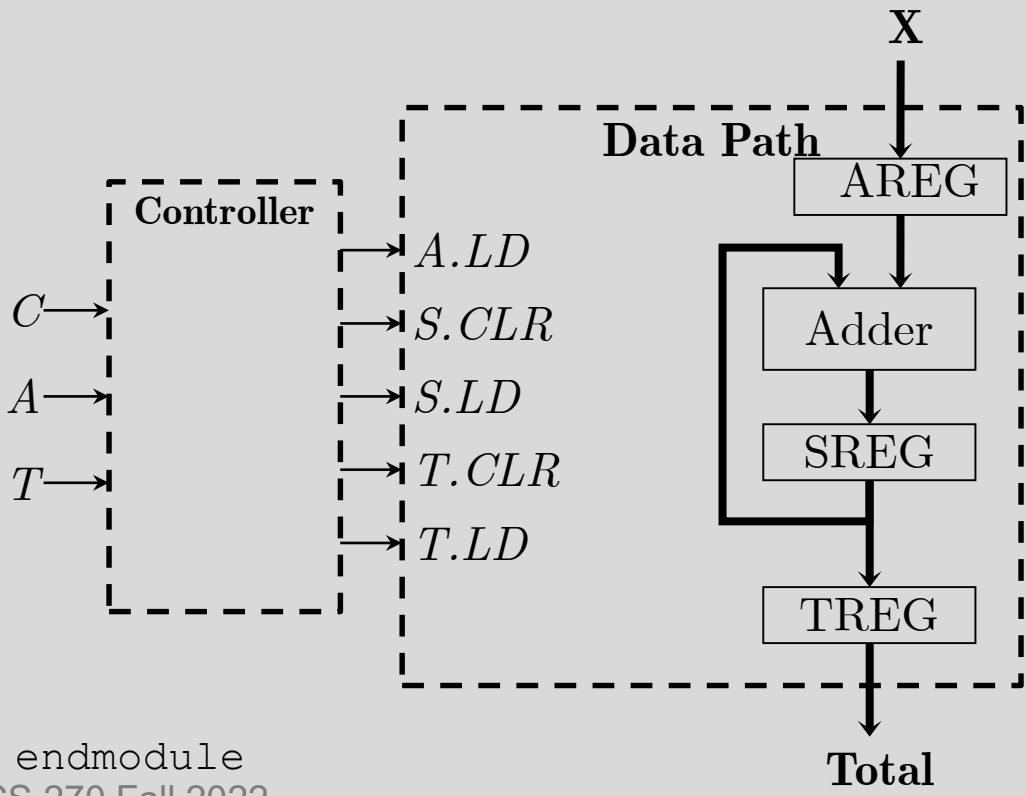
I do not need to keep the value of the AREG at the last X that came in. Because then I would be adding x multiple times.



endmodule

```
module CashReg
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller State Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module CashReg
 always @(posedge Clock)
 begin
   AREG <= A_LD ? X : 'd0; // CORRECT
   SREG <=
     S_CLR ? 'd0 :
       (S_LD ? R : SREG);
   TREG <=
     T_CLR ? 'd0 :
       (T_LD ? SREG : TREG);
 end
```



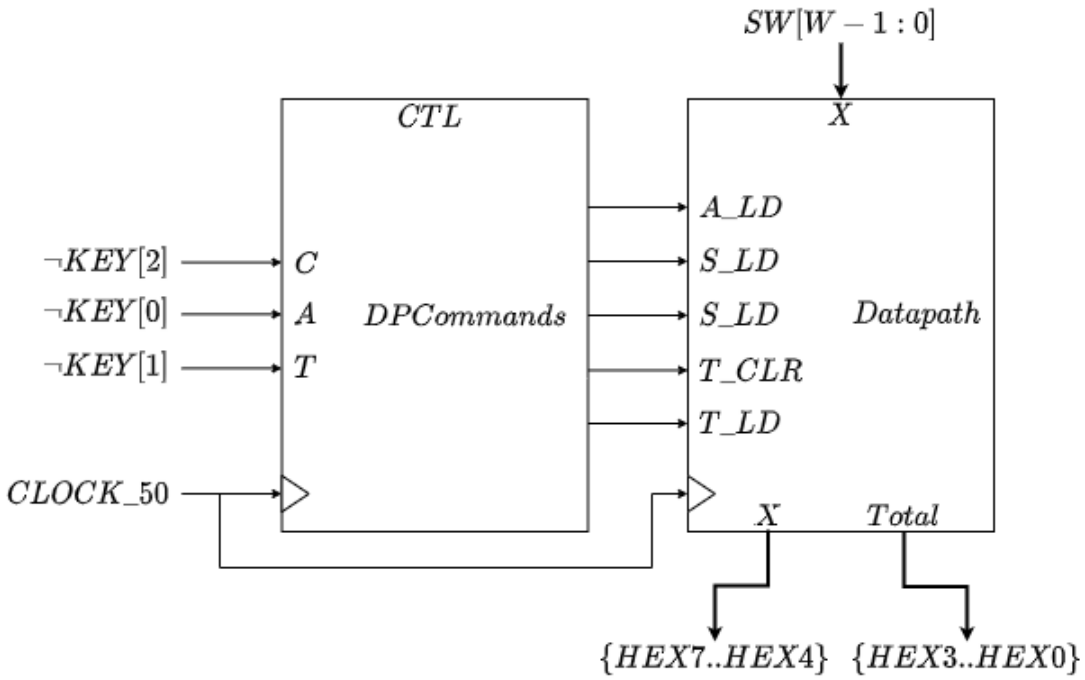

```
endmodule
```

```
module CashReg
// Module inputs and outputs

// Datapath Components

// Datapath Controller

// Controller State Transitions

// Initial State

// Controller State Update

// Controller Output Logic

// Datapath State Update

// Datapath Output Logic

endmodule
```

```
module CashReg
 assign Total = TREG;
 assign QState = Q;
```

```
endmodule
```

Lab 7: DE2-115 Interface




```
module CashReg (Clock, C, A, T, X, Total);
. . .
endmodule

CashRegister(CLOCK_50,SW,KEY,HEX7,HEX6,HEX5,HEX4,HEX3,HEX2,HEX1,HEX0);
    . . .
    CashReg CR(CLOCK_50,~KEY[2],~KEY[0],~KEY[1],SW[W-1:0],Total);
    . . .
endmodule
```

Parameterized Designs


```
// W-bit Ripple-Carry Adder

module AddW #(parameter W = 8)  // Default width
  (A, B, c0, S, ovf);
  input [W-1:0] A, B;           // W-bit unsigned inputs
```

This works with ModelSim
But
not with Quartus!

```
// Instantiate and "chain" W full adders
  genvar i;
  generate
    for (i = 0; i < W; i = i + 1)
      FullAdder FA[W-1:0](A[i], B[i], c[i], S[i], c[i+1]);
  endgenerate

// Overflow
  assign ovf = c[W];
endmodule
```

Parameterized Designs

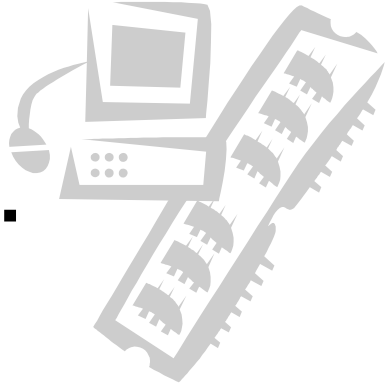
```
// W-bit Ripple-Carry Adder

module AddW #(parameter W = 8)   // Default width
  (A, B, c0, S, ovf);
  input [W-1:0] A, B;              // W-bit unsigned inputs
  input c0;                        // Carry-in
  output [W-1:0] S;                // W-bit unsigned output
  output ovf;                      // Overflow signal

  wire [W:0] c;                    // Carry signals
  assign c[0] = c0;                // Initial carry

// Instantiate and "chain" W full adders
  genvar i;
  generate
    for (i = 0; i < W; i = i + 1)
      FullAdder FA[W-1:0](A[i], B[i], c[i], S[i], c[i+1]);
  endgenerate

// Overflow
  assign ovf = c[W];
endmodule
```

A *Kludgy* Solution for Quartus

```
// W-bit Ripple-Carry Adder

module A                                                    8-bit
  (A, B,
  input
  input
  output
  output

  wire [W
  assign

// Insta
  genvar
  generat
    for
      Ful
  endgene

// Overf
  assign
endmodule
```

```
//
  FullAdder F0(A[0], B[0], c[0], S[0], c[1]);
//
  FullAdder F1(A[1], B[1], c[1], S[1], c[2]);
//
  FullAdder F2(A[2], B[2], c[2], S[2], c[3]);
//
  FullAdder F3(A[3], B[3], c[3], S[3], c[4]);
/*
  FullAdder F4(A[4], B[4], c[4], S[4], c[5]);
//
  FullAdder F5(A[5], B[5], c[5], S[5], c[6]);
//
  FullAdder F6(A[6], B[6], c[6], S[6], c[7]);
//
  FullAdder F7(A[7], B[7], c[7], S[7], c[8]);
*/
```

Ugly solution but it works …



Two Copies of AddW

```
// Quartus top-level module file
module CashRegister(...);
...
endmodule

// Explicit W-bit Adder
module AddW #(parameter W = 8)

//
  FullAdder F0(A[0], B[0], c[0], S[0], c[1]);
//
  FullAdder F1(A[1], B[1], c[1], S[1], c[2]);
...
endmodule
```

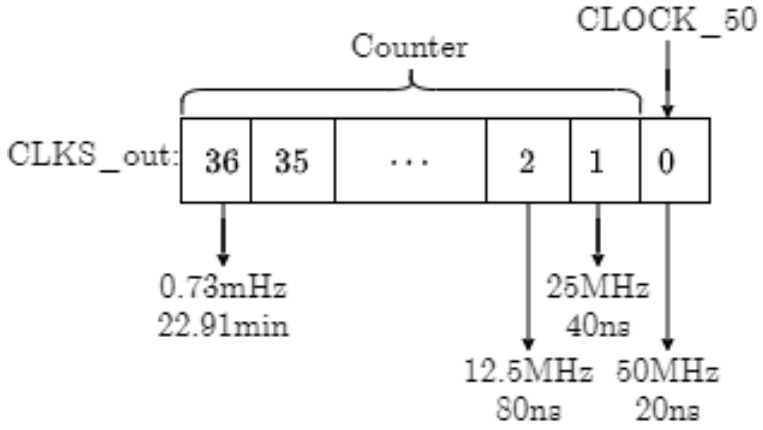
```
// Testbench top-level module file
module Testbench;
...
endmodule

// genvar W-bit Adder
module AddW #(parameter W = 8)
...
genvar i;
  generate
    for (i = 0; i < W; i = i + 1)
...

endmodule
```

相比较于 datapath，Controllar更容易有 bug:(何时执行哪种操作)

ModelSim & Quartus Demo

Getting Timing to Work

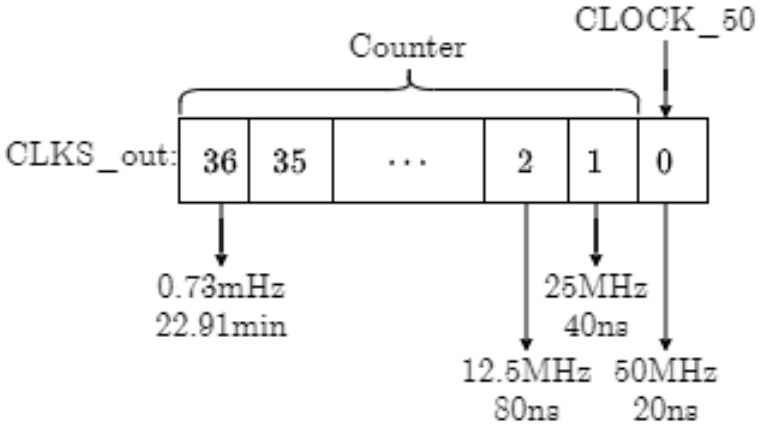
```
module Clock_Div
  #(parameter SIZE = 36)
  (input CLK_in, output [SIZE:0] CLKS_out);

  reg [SIZE:1] Counter;
  initial Counter = 'd0;

  always @(posedge CLK_in)
    Counter <= Counter + 1;

  assign CLKS_out = {Counter, CLK_in};
endmodule
```




```
CashRegister(CLOCK_50,SW,KEY,HEX7,HEX6,HEX5,HEX4,HEX3,HEX2,HEX1,HEX0);
    . . .


    CashReg CR(CLOCK_50,~KEY[2],~KEY[0],~KEY[1],SW[W-1:0],Total);
    . . .
endmodule
```

Getting Timing to Work

```
module Clock_Div
  #(parameter SIZE = 36)
  (input CLK_in, output [SIZE:0] CLKS_out);

  reg [SIZE:1] Counter;
  initial Counter = 'd0;

  always @(posedge CLK_in)
    Counter <= Counter + 1;

  assign CLKS_out = {Counter, CLK_in};
endmodule
```




```
CashRegister(CLOCK_50,SW,KEY,HEX7,HEX6,HEX5,HEX4,HEX3,HEX2,HEX1,HEX0);
    . . .
    wire [36:0] CLOCKS;
    Clock_Div #(.SIZE(36)) CD(CLOCK_50, CLOCKS);
    CashReg CR(CLOCKS[22],~KEY[2],~KEY[0],~KEY[1],SW[W-1:0],Total);
    . . .
endmodule
```

11.92Hz
83.89ms